# Snooze: A Scalable and Autonomic Cloud Management System

Eugen Feller, Matthieu Simonin, Yvon Jégou, Anne-Cécile Orgerie, David Margery, Christine Morin

▶ **To cite this version:**

Eugen Feller, Matthieu Simonin, Yvon Jégou, Anne-Cécile Orgerie, David Margery, et al.. Snooze: A Scalable and Autonomic Cloud Management System. [Research Report] RR-8649, Inria Rennes; INRIA. 2014, pp.31. hal-01091755

**HAL Id: hal-01091755**
**https://inria.hal.science/hal-01091755**

Submitted on 8 Dec 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.


# Snooze: A Scalable and Autonomic Cloud Management System

**Eugen Feller, Matthieu Simonin, Yvon Jégou, Anne-Cécile Orgerie, David Margery, Christine Morin**

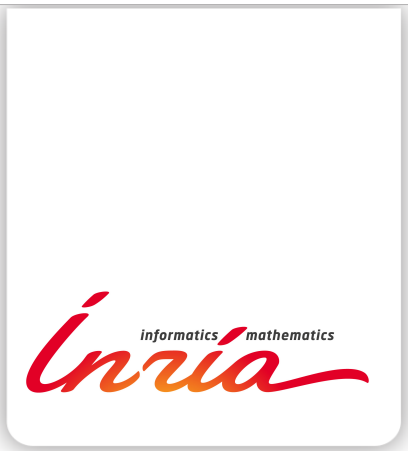
**RESEARCH REPORT**

**N° 8649**

December 2014

Project-Teams MYRIADS

ISSN 0249-6399 ISRN INRIA/RR--8649--FR+ENG




# Snooze: A Scalable and Autonomic Cloud Management System

Eugen Feller *, Matthieu Simonin *, Yvon Jégou *, Anne-Cécile Orgerie †, David Margery *, Christine Morin *

Project-Teams MYRIADS

Research Report n° 8649 — December 2014 — 28 pages

**Abstract:** With the advent of cloud computing and the need to satisfy growing customers resource demands, cloud providers now operate increasing amounts of large data centers. In order to ease the creation of private clouds, several open source and commercial Infrastructure-as-a-Service cloud management frameworks have been proposed during the past years. However, all these systems are either highly centralized or have limited fault tolerance support. Consequently, they all share common drawbacks: scalability and Single Point of Failure. In this paper, we present the design, implementation and evaluation of a novel scalable and autonomic virtual machine (VM) management framework called Snooze. For scalability the system utilizes a self-configuring hierarchical architecture and performs distributed VM management. Moreover, fault tolerance is provided at all levels of the hierarchy, thus allowing the system to self-heal in the event of failures. Our large-scale evaluation conducted across multiple geographically distributed clusters of the Grid'5000 experimentation testbed shows that the system scales to over ten thousand system services and can easily manage hundreds of VMs. Moreover, it is robust enough to automatically recover the hierarchy in the event of thousands of concurrent system services failures.

**Key-words:** Cloud Computing, Virtualization, Infrastructure-as-a-Service, Scalability, Self-Organization, Self-Healing

* INRIA, Rennes, France - {Eugen Feller, Christine Morin, Matthieu.Simonin, Yvon.Jegou}@inria.fr
† CNRS, IRISA Laboratory, Rennes, France - Anne-Cecile.Orgerie@cnrs.fr

**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

# Snooze: un système extensible et autonome de gestion de nuages informatiques

**Résumé :** Avec l'avènement des nuages informatiques et le besoin de satisfaire les demandes de ressources croissantes des utilisateurs, les fournisseurs de nuages gèrent désormais un nombre croissant de grands centres de données. Afin de faciliter la création de nuages privés, plusieurs systèmes de gestion de nuages offrant des services d'infrastructure, open source et commerciaux, ont été proposés ces dernières années. Cependant, tous ces systèmes sont très centralisés ou offre un support limité pour la tolérance aux fautes. Par conséquent, ils partagent plusieurs inconvénients : faible extensibilité et point unique de défaillance. Dans cet article, nous présentons la conception, la mise en oeuvre et l'évaluation d'un nouveau système de gestion de machines virtuelles extensible et autonome appelé Snooze. A des fins d'extensibilité, le système utilise une architecture hiérarchique auto-configurable et gère les machines virtuelles de manière distribuée. En outre, la tolérance aux fautes est assurée à tous les niveaux de la hiérarchie, permettant au système de s'auto-réparer en cas de défaillance. L'évaluation à grande échelle nous avons conduite sur des grappes géographiquement distribuées de la plate-forme d'expérimentation Grid'5000 montre que le système passe à l'échelle avec plus de 10 000 services système et peut facilement gérer des centaines de machines virtuelles. De plus, il est suffisamment robuste pour rétablir la hiérarchie dans le cas de milliers de défaillances simultanées.

**Mots-clés :** Nuage informatique, virtualisation, service d'infrastructure, passage à l'échelle, auto-organisation, auto-réparation

# 1 Introduction

Large-scale data centers enabling today's cloud services are hosting a tremendous amount of servers and virtual machines (VMs). The sheer scale of such data centers imposes a number of important challenges on their cloud management systems. First, cloud management systems must remain scalable with increasing numbers of servers and VMs. Second, at scale server failures become inevitable (e.g., see past Amazon outages). Consequently, cloud management systems must be highly available in order to enable continuous cloud service operation. Finally, at scale server administration becomes a demanding task requiring highly skilled IT experts and thus cloud management systems must be designed to be easily configurable. Such challenges call for scalable autonomic cloud management systems which are self-configuring and self-healing.

We focus on Infrastructure-as-a-Service (IaaS) clouds as they serve as the building block to enable most of the available cloud services, ranging from scalable web deployments to parallel data processing. A number of open-source and commercial IaaS cloud management systems such as CloudStack [29], Nimbus [17], OpenStack [30], OpenNebula [22], VMware vCenter Server [31], and Eucalyptus [23] have been proposed in the past. However, the former four systems are based on centralized architectures thus limiting their scalability. Indeed, in [14] the authors show that VMware does not scale beyond 32 servers and 3000 VMs. Eucalyptus improves the scalability via a hierarchical architecture. However, it is not designed to be self-configuring and self-healing. The lack of self-configuration and self-healing properties complicates the system configuration as it requires manual efforts to construct (resp. reconstruct) the hierarchy during initial deployment (resp. in the event of failures). Ultimately, all the aforementioned systems require active/passive servers to achieve high availability thus wasting resources. The scalability and usability limitations of the open-source cloud stacks have been identified in [33].

To tackle these limitations, we have proposed the Snooze [13] IaaS cloud management system that is designed to scale across thousands of servers. Unlike existing systems, for scalability, ease of configuration, and high availability, Snooze is based on a self-configuring and self-healing hierarchical architecture of system services. The hierarchical architecture enables scalable system management by distributing the server and VM management responsibilities to multiple managers with each one having only a partial view of the system. Fault tolerance is integrated at all levels of the hierarchy and thus the system is able to self-heal and continue its operation despite system service failures. No active/passive servers are required to achieve high availability.

In contrast to our previous published work [13], in this paper we give a more detailed description of Snooze and present important aspects of its implementation. Moreover, we conduct an extensive scalability study of Snooze across over 500 geographically distributed servers of the Grid'5000 experimentation testbed [11]. More precisely, we evaluate the Snooze self-configuring and self-healing hierarchy with thousands of system services. We also demonstrate the application deployment scalability across hundreds of VMs on the example of a Hadoop MapReduce application. To the best of our knowledge this is the first evaluation of the scalability of an autonomous cloud management system.

The remainder of this paper is organized as follows. We present the Snooze design principles in Section 2. We describe the Snooze system architecture in Section 3. In Section 4, we detail the hierarchy management protocols. In Section 5, the VM management mechanisms are described. We discuss important implementation aspects in Section 6. In Section 7, the evaluation results are analyzed. We review the related work in Section 8 and present conclusions and future work in Section 9.

# 2 Design Principles

Our goal is to design and implement a scalable and autonomic IaaS cloud management system. Several properties have to be fulfilled by the cloud management system in order to achieve these goals. First, the cloud management system architecture has to scale across many thousands of servers and VMs. Second, the cloud management system configuration requires highly skilled IT experts. Consequently, an easily configurable system is desirable. Third, servers and thus framework management components can fail at any time. Therefore, a cloud management system needs to be able to self-heal and continue its operation despite component failures. In order to achieve the latter two properties, the cloud management system must satisfy the self-configuration and healing properties of an autonomic system [18].

To achieve scalability and autonomy we have made the key design choice to design a system based on a self-organizing and healing hierarchical architecture. Our scalability and autonomy design choices are motivated by previous works which have proven that hierarchical architectures can greatly improve the system scalability. Particularly, the Snooze architecture is partially inspired from the Hasthi [24] autonomic system, which is shown to scale up to 100 000 resources by simulation. However, in contrast to Hasthi whose design is presented to be system agnostic and utilizes a Distributed Hash Table (DHT) based on a Peer-to-Peer (P2P) network, Snooze follows a simpler design and does not require the use of a P2P technology. Moreover, it targets virtualized systems and thus its design and implementation are driven by the system specific objectives and issues.

Organizing the system hierarchically improves its scalability as components at higher levels of the hierarchy do not require global knowledge of the system. The key idea of our system is to split the VM management tasks across multiple independent autonomic managers with each manager having only a partial view of the data center. Particularly, each manager is only in charge of managing a subset of the data center servers and VMs. A coordinator oversees the managers and is contacted by the clients to submit VMs. Once VMs are submitted, clients communicate directly with the managers to control VM life-cycle (e.g., suspend, shutdown) and retrieve VM monitoring information.

The self-configuration and healing properties of the system enable to automate the hierarchy construction at boot time and recovery in the event of failures. More precisely, the coordinator is automatically elected among the managers during the managers startup and in the event of a coordinator failure. Servers automatically discover the current coordinator and are assigned to a manager. In the event of a coordinator failure, an existing manager is promoted to a coordinator. In the event of a manager to coordinator promotion and/or manager failure, servers are automatically reassigned to a new manager. These mechanisms significantly ease the system configuration efforts and ensure continuous system operation despite server failures.

# 3 System Architecture

This section presents the Snooze system architecture. First, the assumptions and system model are introduced. Then, a high-level system overview is presented.

## 3.1 Assumptions and Model

We assume a data center whose servers are interconnected with a high-speed LAN such as Gigabit Ethernet or Infiniband. Multicast support is assumed to be available at network level. Servers can be either homogeneous or heterogeneous. They are managed by a virtualization solution such as Xen [10] or KVM [19]. VMs are seen as black-boxes thus no application-specific knowledge

is required to guide the VM management mechanisms decisions. We assume that failures that partition the network are not tolerated. We assume that simultaneous server failures may happen.

In a virtualized data center multiple VMs are typically collocated on a server. Despite the resource isolation properties of modern virtualization solutions performance isolation is not always guaranteed. In other words, collocated VMs with correlated resource demands (e.g., memory bound VMs) can experience a performance degradation as they typically share the same hardware subsystems (e.g., last level cache). In this work we assume that performance isolation is provided by the underlying virtualization solution. Consequently, implemented VM management algorithms do not take into account complementaries between the VM resource demands during their decision making processes.

Finally, applications inside VMs can incur dependencies on the VMs. For instance, a web server VM will typically have a dependency on an application server VM and/or a database VM. Similarly, security obligations might prevent two VMs from being close to each other. In this work we assume independent VMs. In other words, the integrated VM management algorithms do not support the specification of VM collocation or anti-collocation constraints which could allow (resp. restrict) certain VMs to be collocated. Nevertheless, given that Snooze is flexible enough to support any VM management algorithm nothing prevents the system from integrating such algorithms in the future.

## 3.2 High-level Overview

The high-level overview of the hierarchical Snooze architecture is shown in Figure 1. It is partitioned into three layers: computing, management, and client. At the computing layer, servers

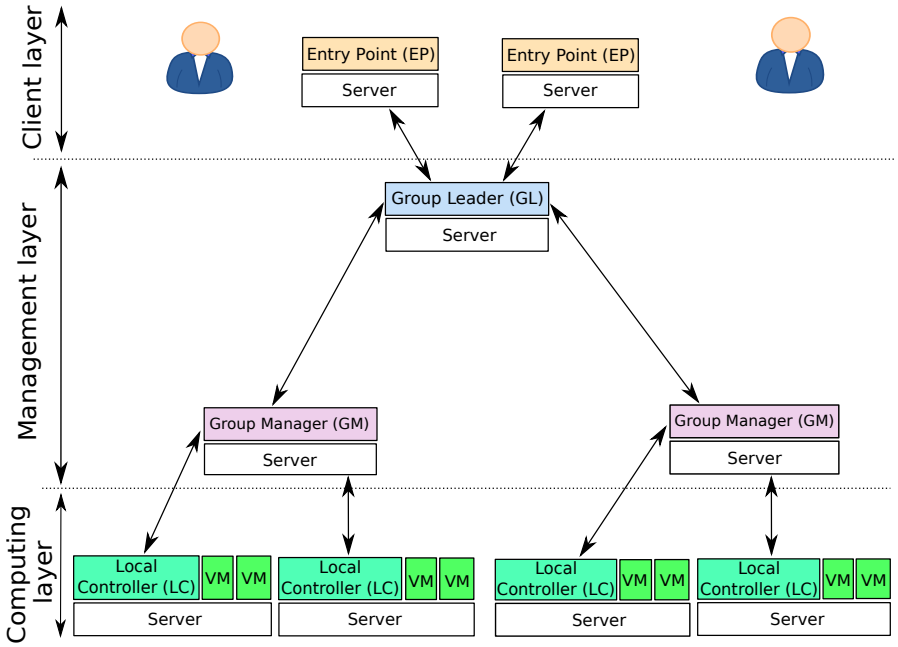



Figure 1: Snooze high-level system architecture overview.

are organized in a cluster in charge of hosting the VMs. Each server is controlled by a system service, the so-called Local Controller (LC). The management layer allows to scale the VM

management system. It is composed of servers hosting fault-tolerant system services: one coordinator, the so-called Group Leader (GL) and one or more autonomic managers, the so-called Group Managers (GMs). System services are organized hierarchically. GL oversees the GMs. It is elected among the GMs during the hierarchy self-configuration and in the event of a GL failure. Each GM manages a subset of LCs and VMs. The GL receives VM submission requests from the clients and distributes them among the GMs. GMs place VMs on LCs. Once VMs are submitted, clients interact directly with the GMs to control the VMs life-cycle (e.g., shutdown, reboot). As the GL can change over the time, a method is required in order for the clients to discover the current GL. This functionality is provided by the client layer. The client layer is composed of a predefined number of services, the so-called Entry Points (EPs) which remain updated about the current GL. All system services are accessible through a RESTful interface. Consequently, any client software (e.g., Command Line Interface (CLI), web, Cloud Library) can be used to interact with the EPs, GL, and the GMs.

Given a set of servers it is up to the system administrator to decide on the number of LCs and GMs upon Snooze deployment. For instance, in the most basic deployment scenario two GMs and one LC are required. One of the GMs will be promoted to a GL as part of a leader election procedure. Note, that system services have dedicated roles. Consequently, a newly elected GL is no longer a GM and thus does not provide GM services. System services are flexible enough to co-exist on the same server. For instance, it is possible to collocate a GM and LC system service on the same server.

### 3.2.1 Local Controllers

Each LC enforces VM life-cycle commands coming from its assigned GM. Examples of such commands include VM start, reboot, suspend, and shutdown. A LC also monitors VMs and periodically sends VM resource utilization data to its assigned GM. Each LC maintains a repository with information (e.g., status, assigned IP) about the currently running VMs on its server.

### 3.2.2 Group Managers

Each GM is in charge of the management of a subset of the LCs. It receives VM resource utilization data from LCs and stores it in a local repository. Based on this data the GM estimates VM resource utilization and takes VM placement decisions. VM placement mechanisms are triggered event-based to handle VM submission requests arriving from the GL.

GM summary information is periodically sent by each GM to the current GL in order to support high-level VM to GM distribution decisions (see Section 5.1 for more details on monitoring). Finally, GMs are also contacted by the client software to control VMs life-cycle and retrieve VM information (e.g., resource utilization, status).

### 3.2.3 Group Leader

The GL manages the GMs. It is in charge of assigning LCs to GMs, accepting clients VM submission requests, managing VM networking and dispatching the submitted VMs among the GMs. Moreover, it receives GM summary information and stores it in a repository. We now discuss these tasks in more detail.

When LCs join the hierarchy either initially or in the event of a GM failure, they need to get a GM assigned. LC to GM assignment decisions are guided by a LC assignment policy. For example, LC can be assigned to GMs in a round-robin fashion or based on the current GM utilization. Once the LCs are assigned to GMs, VMs can be submitted by the clients to the GL. In case client VM submission requests arrive before LCs are assigned to GMs, an error message

is returned. Note, that additional LCs can join the system at any time without disturbing its normal functioning.

Once a group of VMs (one or multiple VMs) is submitted to the GL, VM to GM dispatching decisions are taken by the GL. They are implemented using a VM dispatching policy which decides on the assignment of VMs to GMs. In order to compute this assignment the GM summary information is used, which contains information about aggregated GM resource utilization. Based on this information VM to GM dispatching decisions are made. However, before a VM can be dispatched to GMs, VM networking needs to be managed in order for the VMs to become reachable to the outside world after its startup. This process involves two steps: (1) getting an IP address assigned to the VM; (2) configuring the network interface based on the assigned IP. The GL is in charge of the former step. Therefore, it maintains a system administrator configurable subnet from which it is allowed to assign IP addresses. When VMs are submitted to the GL, each of them automatically gets an IP address assigned from this subnet. The assigned IP address is embedded in the VMs MAC address. When the VM boots it decodes the IP from its MAC address and performs the network configuration.

The GL does not maintain a global view of the VMs in the system. Instead, after the VM dispatching, information about the GMs on which the VMs were dispatched is stored on the client-side thus allowing the clients to directly interact with the corresponding GMs for subsequent VM management requests. Despite the lightweight VM dispatching decisions and no global view of the VMs, the GL scalability can be further improved with replication and a load balancing layer.

#### 3.2.4 Entry Points

In Snooze, the GL is automatically elected among the GMs during the system startup and in the event of a GL failure. Consequently, a GL can change over time. In order for the clients to instruct a GL to start VMs, they must be provided a way to discover the current GL. In order to achieve this we introduce a predefined number of EPs. EPs are system services which typically reside on servers of the same network as the GMs and keep track about the current GL. EPs are contacted by the clients whenever they need to submit VMs to discover the current GL.

# 4 Hierarchy Management

This section describes how the Snooze system hierarchy is constructed and maintained. First, the heartbeat mechanisms are introduced. Then, the self-configuration and healing mechanisms are presented. Self-configuration refers to the ability of the system to dynamically construct the hierarchy during startup. On the other hand, self-healing allows to automatically reconstruct the hierarchy in the event of system service or server failures.

## 4.1 Heartbeats

To support self-configuration and healing, Snooze integrates heartbeat protocols at all levels of the hierarchy.

The GL periodically sends its identity to a dedicated GL heartbeat multicast group containing all EPs and GMs. GL heartbeats allow the EPs to remain updated about the current GL. GL heartbeats are also required by the LCs and GMs to discover the current GL during boot time and in the event of a GM (resp. GL) failure. More precisely, the current GL needs to be discovered by the LCs during boot time and in the event of a GM failure in order to get a GM assigned.

The GMs need to discover the current GL during boot time and in the event of a GL failure in order to register with the GL.

The GMs periodically announce their presence to the assigned LCs on a per-GM defined heartbeat multicast group. LCs use GM heartbeats to detect a GM failure.

In order for the GL and GMs to detect GM (resp. LCs) failures unicast-based heartbeats are used. They are piggybacked by the GMs and LCs with their monitoring data which is periodically sent to the GL (resp. GM).

## 4.2 Self-Configuration

When a system service is started on a server it is statically configured to become either a LC or a GM. When the services boot, the first step in hierarchy construction involves the election of a GL among the GMs. After the GL election, other GMs need to register with it. For a LC to join the hierarchy, it first needs to discover the current GL and get a GM assigned by contacting the GL. Once it is assigned to a GM, it can register with it. We now describe all these steps in more details.

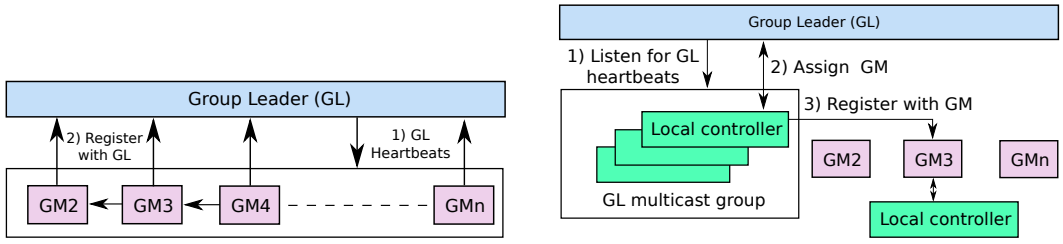
### 4.2.1 Group Leader Election and Group Manager Join

When a GM starts, the GL election algorithm is triggered. Currently, our GL election algorithm is built on top of Apache ZooKeeper [15], a reliable and scalable distributed coordination system. It follows the recipe proposed by the ZooKeeper community in [4]. The key idea of the algorithm is to construct a chain between the GMs in which each GM watches its predecessor GM. The GM with the lowest Unique Identifier (UID) becomes the GL. The GL election works as follows. Each GM contacts the ZooKeeper service upon startup to get a UID and have the ZooKeeper service create an entry associating the GM UID with its network address. The ZooKeeper service guarantees that the first GM contacting it receives the lowest UID. After the entry creation each GM first attempts to find an entry in the ZooKeeper service with the next lowest UID. To do so, a list of entries is retrieved by leveraging the ZooKeeper API and sorted by the GM in decreasing order. If an entry with a lower UID exists, the GM starts watching it and joins the GL heartbeat multicast group to get informed about the current GL, otherwise it becomes the GL and starts announcing its presence by sending GL heartbeat messages on the GL heartbeat multicast group.

Once the GMs which were not promoted to be a GL receive a GL heartbeat message, they register with the GL by sending their UID and network address. This process is shown in Figure 2a.

### 4.2.2 Local Controller Join

The join process of a LC works as follows. Each time a LC starts it subscribes to the GL heartbeat multicast group in order to discover the current GL. When a GL heartbeat message arrives, the actual join process is started by sending a GM assignment request to the GL. The GL distinguishes between two scenarios. In the first scenario a LC joins the system as part of its usual boot process. In the second scenario a LC joins the system after a maintenance and/or failure period. To support the former scenario the LC can be assigned to any GM. However, to enable the latter scenario, the LC should be preferably assigned to its previous GM in order to mitigate hierarchy imbalance. In order to enable a LC rejoin, when a GL receives a LC to GM assignment request it queries the GMs for the status of the joining LC. In case any GM replies to the GL that it was previously in charge for the joining LC, the GL returns the network address of the GM to the LC, which then initiates the actual GM registration process by sending its description (i.e., UID, network address, available capacity) to the GM (see Figure 2b). Finally,




(a) Group manager join procedure.

(b) Local controller join procedure.

Figure 2: Self-configuration: GM and LC join procedures. (a) GM discovers the GL via heartbeat and registers with the GL by sendings its contact information. (b) LC joins the GL multicast group to discover the current GL. The GL is then contacted to get a GM assigned. Once the GM is assigned, the LCs sends its description.

the LC unsubscribes from the GL heartbeat multicast group and starts listening for its assigned GM heartbeat messages to detect GM failures. Moreover, it periodically sends VM resource utilization data to its assigned GM.

In case none of the GMs was previously in charge for the LC, the GL triggers the system administrator selected GM assignment policy which assigns the LC to a GM according to its high-level objective (e.g., round robin). The network address of the assigned GM is returned to the LC and the LC initiates the GM registration process by sending its description.

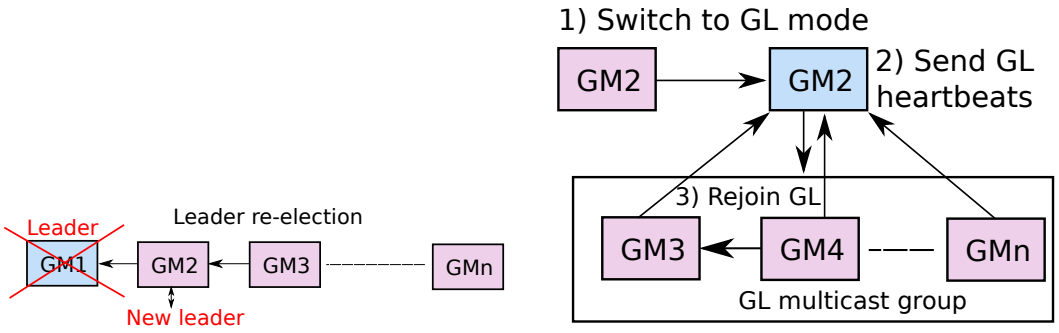
The aforementioned mechanisms allow Snooze to provide autonomy via self-configuration thus significantly reducing the system configuration efforts. The hierarchy is then constructed fully automatically without human intervention. The system administrator only has to specify for each server if it will act as a LC or GM.

## 4.3 Self-Healing

Self-healing is performed at all levels of the hierarchy. It involves the detection of and recovery from LC, GM, and GL failures.

**Local controller failures** are detected by their assigned GM based on a unicast heartbeat timeout. Once a LC failure is detected, the GM gracefully removes the failed LC from its repository in order for it not to be considered in future VM management tasks. Note, that in the event of a LC failure, the VMs it hosts are terminated. IP addresses assigned to the terminated VMs must be recycled by the GL in order to avoid IPs leakage. Therefore, IPs of the terminated VMs are added to a list of recyclable addresses which exists on each GM. This list is periodically sent to the GL. Upon reception of the terminated VM IPs, the GL adds the IPs back to its pool of managed addresses. Note, that for the time being Snooze does not handle the recovery of terminated VMs. However, snapshot features of hypervisors can be used by LCs in order to periodically take VM snapshots. This will allow a GM to reschedule the failed VMs on its remaining active LCs.

**Group manager failures** are detected by the GL and LCs based on unicast (resp. multicast) heartbeat timeouts. When a GM fails all its knowledge (e.g., VMs and their assigned IP addresses, LC network addresses) is lost and the GM is removed from the GL repository in order to prevent the GL from sending VM submission requests to dead GMs. LCs which were previously managed by the failed GM start the rejoin procedure. Similar to the join procedure a rejoin involves the assignment of a GM to the LC. However, during the rejoin, LCs knowledge about the currently running VMs must be transferred to the newly assigned GM. Otherwise,




(a) Group leader failure.

(b) Group leader switch.

Figure 3: Self-healing: GL failure and GL switch examples. (a) A GL failure is detected by the predecessor GM based on timeout in the Apache ZooKeeper service. (b) The predecessor GM becomes the new GL by giving up its GM responsibilities and starting sending GL heartbeats. Other GMs receive the GL heartbeat and rejoin the hierarchy.

clients will fail to manage their VMs (see Section 6.2 for more details). Finally, as every GM is being watched by its successor GM in the ZooKeeper service, a failure event is triggered on the successor GM. Upon a GM failure the successor GM relies on the Apache ZooKeeper service in order to discover a new predecessor GM and starts watching it. This is achieved by triggering the previously described GM join procedure (see Section 4.2.1).

**Group leader failure** is detected by its successor GM based on a timeout triggered by the ZooKeeper service. When a GL fails, first a new GL must be elected among the GMs. The newly elected GL then must be discovered and joined by the remaining GMs. Moreover, LCs which were previously assigned to the GM becoming the new GL must rejoin the hierarchy. Note, that in the event of a GL failure all its knowledge about the existing GMs as well as the VM networking information (i.e., assigned IP addresses) is lost. Consequently, this knowledge must be rebuilt in order for the system to remain in a consistent state. These steps are achieved as follows.

Upon a GL failure, the successor GM in the chain becomes the new GL as its UID is the next lowest one (see Figure 3a). When a GM is promoted to be a GL, it gracefully terminates all its tasks such as the heartbeat and summary information sending. Moreover, it clears its internal LC and VM knowledge. Afterwards, the GM switches to GL mode, and starts sending its network address to the GL heartbeat multicast group. The switch to GL mode yields two effects: (1) the newly elected GL must be detected and joined by the remaining GMs; (2) LCs which were previously managed by the GM must rejoin the hierarchy to another GM as the GL no longer plays the role of an ordinary GL.

In order to detect the new GL, GMs always keep listening for GL heartbeat messages. Upon reception of a new GL network address they trigger the GL rejoin procedure (see Figure 3b). In contrast to the previously introduced GM join procedure, a GM rejoin requires additional data to be sent. Particularly, in order to rebuild GLs VM networking knowledge, each time a GM registers with a new GL it attaches the IPs of its active VMs to its description thus allowing the GL to reconstruct its view of already assigned IPs. Moreover, the GM summary information is periodically sent back to the new GL (see Section 5.1 for more details) thus allowing it to rebuild its GM resource utilization knowledge.

Finally, as a GM has been promoted to become the new GL it stopped sending heartbeat messages to its heartbeat multicast group. LCs which were previously assigned to it thus fail to

receive the heartbeat messages and consider it as failed. They rejoin the hierarchy by following the GM failure recovery procedure.

The aforementioned mechanisms enable Snooze to satisfy the self-healing property of autonomic computing systems. System services failures are automatically detected and the hierarchy is restored in order to continue normal functioning without external intervention.

# 5 VM Management

We now present the core VM management mechanisms of Snooze. We focus primary on the resource utilization monitoring mechanisms as these mechanisms are likely to impact the system scalability. For the sake of self-containedness we also briefly describe the VM dispatching and placement mechanisms.

## 5.1 Resource Utilization Monitoring

In order to support VM dispatching and placement decisions, resource utilization monitoring is performed at all layers of the system while considering four dimensions: CPU, memory, network Rx and Tx. At the computing layer, VMs are monitored and VM resource utilization information is periodically reported by the LCs to their assigned GM. At the management layer, GMs periodically send summary information to the GL.

VMs are assigned with four-dimensional requested and used capacity vectors. Requested capacity reflects the VM resource requirements at submission time. It includes the number of requested cores, memory size, and the Rx/Tx network capacity. The used capacity vector corresponds to the estimated VM resource utilization at a given time. The VM used capacity is computed as follows. First, a predefined number of monitoring entries is collected. A monitoring entry is a four-dimensional vector which includes the current VM CPU load, memory consumption, and Rx/Tx network utilization. VM CPU load is computed as a percentage of the occupied server CPU time over a measurement interval, memory is measured in KiloBytes and network utilization in Bytes/sec. Based on the collected monitoring entries, a used capacity vector is computed. VM used capacity can be either computed by simply considering the average of the $n$ most recent monitoring entries for each resource. Alternatively, more advanced algorithms (e.g., based on Autoregressive-Moving-Average) can be used. In this work the former approach is taken by the LC. VM used capacity vectors are computed for all VMs on behalf of an LC. Once the VM used capacity vectors have been computed they are send by the LC to the assigned GM.

Servers at the computing layer are assigned with four-dimensional total, requested, and used capacity vectors. The total capacity vector represents the total amount of server resources. The requested capacity represents the total amount of capacity requested by the VMs. The requested capacity vector is computed by summing up the VM requested capacity vectors. The used capacity vector represents the server's current resource utilization. It is computed by summing up the VM used capacity vectors.

GM summary information includes the aggregated resource utilization of all servers managed by a GM. Aggregated resource utilization captures the total, requested, and used capacity of a GM. The GM total, requested, and used capacity vectors are computed periodically by summing up the servers total, requested, and used capacity vectors.

## 5.2 VM Dispatching and Placement

When a client attempts to submit VMs to the GL, a dispatching algorithm is used to distribute the VMs among the GMs. For example, VMs can be dispatched in a Round-Robin or First-Fit

fashion. A dispatching algorithm takes as input the submitted VMs and the list of available GMs including their associated aggregated resource utilization data and outputs a dispatching plan which specifies the VM to GM assignments. Note, that aggregated resource utilization is not sufficient to take exact dispatching decisions. For instance, when a client submits a VM requesting 2GB of memory and a GM reports 4GB available it does not necessary mean that the VM can be finally placed on this GM as its available memory could be distributed among multiple LCs (e.g., 4 LCs with each 1 GB of RAM). Consequently, a list of candidate GMs is provided by the VM dispatching policy for each VM. Based on this list, a linear search is performed by the GL during which it sends VM placement requests to the GMs.

Once the VMs have been assigned to the GMs, a VM placement request is sent to the GMs in order to place the VMs on servers. Note that in the implementation of Snooze the requests sent to the GMs are processed concurrently (Section 7.4 gives an evaluation of the submission mechanisms). Once a GM receives a VM placement request, it triggers a VM placement algorithm to compute a VM to server allocation. The algorithm takes as input the VMs to be assigned and lists of available servers including their total, requested, and used capacity vectors. Similarly, to VM dispatching, VMs can be placed either in a Round-Robin or First-Fit fashion. Moreover, thanks to the integrated resource utilization monitoring, VM dispatching and placement algorithms can base their decisions on either the GM (resp. server) requested or used capacity vectors. Finally, often VMs need to be sorted by the VM dispatching and placement algorithms. Sorting vectors requires them to be first normalized to scalar values. Snooze supports a number of sorting norms such as L1, Euclid or Max. In this work the L1 norm [32] is used.· It is up to the system administrator to choose which algorithms, vectors, and sort norms will be used to perform VM dispatching and VM placement during the system configuration.

# 6 Implementation

We have developed a Snooze prototype from scratch in Java. It currently comprises of approximately 15 000 lines of highly modular code. This section presents a few but important implementation aspects. They involve the VM life-cycle enforcement and monitoring, command line interface (CLI), asynchronous VM submission processing, and repositories.

## 6.1 VM Life-Cycle Enforcement and Monitoring

Different tools can be used to control the VM life-cycle management as well as to collect monitoring information. For example, life-cycle management commands can be either enforced by directly contacting the hypervisor API or using an intermediate library such as libvirt [26], which provides a uniform interface to most of the modern hypervisors. Similarly, VM resource utilization data can be obtained from different sources (e.g. libvirt, Ganglia [2], Munin [3]). Snooze provides abstractions to integrate any VM life-cycle management and monitoring solution but currently relies on libvirt. Libvirt is particularly interesting as it can be used for both, VM life-cycle management and transparent VM resource utilization monitoring. This way dependencies on third party tools (e.g. Ganglia) can be minimized.

## 6.2 Command Line Interface

A Java-based CLI is implemented on top of the RESTful interfaces exported by Snooze system services. The CLI supports the definition and management of virtual clusters (VCs). VCs are constructed on the client side to represent collections of one or multiple VMs. VCs are used by the CLI to group VMs and perform collective VM commands (e.g., start, reboot, suspend).

Moreover, visualizing and exporting the current hierarchy organization in the GraphML format is supported by the CLI.

When a user attempts to start a VC, the CLI first transparently discovers an active EP by walking through the EPs list specified in its configuration file. This list must be statically configured during the CLI setup. Given that an active EP exists, a GL lookup is sent in order to receive the current GL network address. Finally, the request to submit the VC is delegated to the GL which dispatches the VMs on the available GMs. The response provides the following information: assigned VM IP addresses, network addresses of the GMs managing the VMs, VM status and an error code which indicates problems during the submission (e.g., not enough resources, bad path to the VM disk image). The GM network addresses on which the VMs were dispatched are stored in the CLI repository, thus allowing the CLI to directly contact the GM whenever VC/VM management commands need to be performed.

Finally, it is important to mention that in the event of GM failures the CLI repository information becomes obsolete. When the CLI attempts to contact a GM managing a given VM which is not reachable, it queries the EP in order to discover the current GL. Afterwards, a GM discovery request is sent to the GL. Upon reception of the request, the GL queries the currently active GMs in order to find the one assigned to the VM, and returns the result to the CLI. In the event of a LC failure all the knowledge about the LC including its VMs is removed from the GM. When a client contacts a GM which does not have knowledge about a certain VM it responds to the client with the appropriate error code.

## 6.3 Asynchronous VM Submission Processing

In order to stay scalable with an increasing number of client requests, VM submissions are processed asynchronously by the GL as well as the GMs. When any client software attempts to submit VMs to the GL, it receives a task identifier (TID) and is required to periodically poll the GL for the response (i.e. long-polling design pattern). VM submission requests are queued on the GL and processed sequentially. For each VM submission request, the GL performs the VM dispatching on behalf of the client by instructing the GMs to place the VMs. Thereby, it receives a TID from each of the GMs upon VM placement request submission and polls the GMs for responses. GMs perform the VM placement tasks on behalf of the GL. When a GL collects responses from all GMs it associates a submission response with the client TID and the request is considered as finished.

Each GM implements a state machine which starts rejecting VM submission requests when it becomes busy. A GM is considered busy when it already processes VM management requests. The GL periodically retries sending VM placement requests to GMs according to the system administrator specified interval and number of retries. An error code is associated with the client request by the GL if after this period no VMs could be placed. This way infinite loops are avoided.

## 6.4 Repositories

Each system service implements a repository for data storage. For example, the GL stores GM descriptions, GM resource utilization and VM networking information. Each GM maintains a local view of its managed LCs and their associated VM resource utilization data. LCs store information about the currently running VMs. Snooze is not limited to a particular repository implementation. Consequently, different storage backends (e.g. MySQL [1], Apache Cassandra [20], MongoDB [9]) can be integrated. However, Snooze currently relies on an in-memory storage based on a ring buffer. In other words, the repository keeps a limited amount of data and

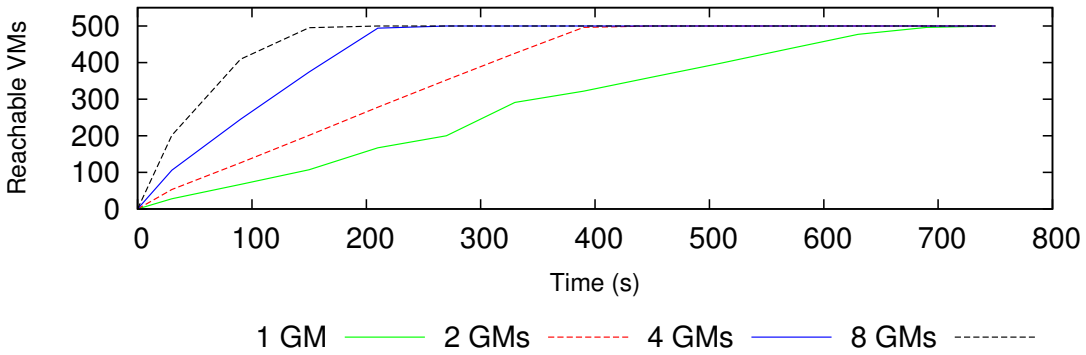



Figure 4: VMs submission time for 500 VMs with an increasing number of GMs. As it can be observed, VMs become reachable faster with more GMs due to parallelization.

starts overwriting the least recently used data once the limit is reached. It is up to the system administrator to set the appropriate limit during the Snooze deployment for each GM.

# 7 Evaluation

This section presents an evaluation of the Snooze system. First, the platform setup and the evaluation criteria are introduced. Then, the experimental results are presented and analyzed.

## 7.1 Platform Setup

We have deployed Snooze in a heterogeneous environment across five geographically distributed sites of the Grid'5000 experimentation testbed. The clusters are located in Rennes, Sophia, Nancy, Lyon, and Toulouse (France). They are interconnected using 10 Gigabit Ethernet links backed by the Renater [7] networking infrastructure. Gigabit Ethernet is used within the clusters. To create the illusion of all the servers belonging to the same cluster, a global Virtual LAN (VLAN) is set up. A maximum of 538 servers are used in our experiments. The exact server hardware specifications are not relevant for the type of experiments performed in this work and thus are neglected due to space restrictions. All servers are running the Debian Wheezy operating system and KVM is used as the virtualization technology. libvirt v0.9.2 serves as the virtualization API. Moreover, Apache Zookeeper v3.3.5 is used to coordinate the leader election. Given the limited number of servers, to evaluate the scalability of the system we deploy multiple system services per server. This allows us to deploy and run several thousands of system services thus resulting in a large hierarchy. The Snooze parameters are configured as follows. We have set the GM and LC heartbeat and monitoring intervals to 3 seconds. The in-memory repository ring-buffer sizes at the GL and GM are set to 20 and 30, respectively. Note that parameter tuning is known to be a non-trivial task for any system. In this work, the parameters are set according to our experience. LC assignment, VM dispatching, and VM placement are done in a round robin fashion.

| Topology / Time | GL scalability<br>1 GL,<br>5K GMs | GM scalability<br>1 GL, 1 GM,<br>5K LCs | System Scalability<br>1GL, 1K GMs,<br>10K LCs |
|---|---|---|---|
| 30 seconds | 1485 GMs | 509 LCs | - |
| 1 minutes | 3861 GMs | 1043 LCs | 979 GMs - 482 LCs |
| 3 minutes | 4656 GMs | 2520 LCs | 983 GMs - 1492 LCs |
| 10 minutes | 4689 GMs | 2633 LCs | 1K GMs - 7436 LCs |
| 15 minutes | 4645 GMs | 4283 LCs | 1K GMs - 9593 LCs |
| 20 minutes | 4629 GMs | 4300 LCs | 1K GMs - 10K LCs |

Table 1: GL, GM, and the system scalability. Snooze scales up to 4629 GMs per GL and 4300 LCs per GM. In total Snooze can manage at least 10000 system services.

## 7.2 Evaluation Criteria

Our evaluation targets the following five key aspects of the system: (1) system setup time; (2) scalability of the VM submission mechanisms; (3) scalability of the self-configuration; (3) system services resource consumption; (4) scalability of the self-healing mechanisms; (5) application deployment scalability impact of the self-healing mechanisms on the application performance.

Evaluating the system setup time is important as it highlights the scalability of the deployment process used to get a Snooze cluster running. Scalability of the VM submission mechanisms shows the VM submission time. It is especially important for the users as it demonstrates the reactivity of the system to serve VM submission requests. The scalability evaluation of the self-configuration and self-healing mechanisms is especially important as it demonstrates the upper bound on the total number of system services the current Snooze prototype can manage as well as how long it takes to recover from a large number of system services failures. The system services resource consumption demonstrates how the resource consumption scales with increasing number of system services and VMs thus giving more insights in the overall scalability of the system. The application deployment scalability shows based on a data analytics application that Snooze can be used to successfully run realistic workloads. Ultimately, the impact of the self-healing mechanisms on the application performance evaluation demonstrates how the application performance is impacted in the presence of a large number of system services failures. This is especially important to evaluate as GL and GM failures require hierarchy reconstruction and thus imply network traffic which can potentially impact application performance.

## 7.3 System Setup Time

The system setup time is composed of two steps: (1) deployment of a Debian OS image; (2) Snooze package installation, configuration and start. The deployment of the Debian OS image is done using the Kadeploy3 [16] provisioning tool. Kadeploy3 leverages a tree-based image dispatching model and thus is able to perform the deployment in a highly scalable manner. The Snooze system services installation, configuration, and start are supported by the parallel remote command execution tool called TakTuk [12]. TakTuk leverages an adaptive work-stealing algorithm to achieve scalable broadcast of commands across a large number of servers.

The largest Snooze deployment with dedicated servers involves 100 GMs and 438 LCs across 538 servers and took 45 minutes. The deployment of the Debian base image takes approximately 30 minutes. The Snooze system service installation and configuration take 15 minutes.

## 7.4 Scalability of the VM Submission Mechanisms

In this section we show how the VM submission time is impacted by the number of GMs. To conduct the experiment, we deploy Snooze with 1 GL, 1000 LCs and vary the number of GMs from 1 to 8 (excluding the GL). For these experiments, the GL and the GMs are deployed on dedicated servers whereas the LCs are deployed on 200 servers each hosting 5 LC system services. 500 VMs are submitted to the system. VM disk images are copied to the servers before VM submission using the SCP Tsunami [8] tool. SCP-Tsunami leverages a BitTorrent protocol to enable scalable propagation of large files across thousands of servers.

Figure 4 depicts the number of VMs reachable after a certain amount of time since the submission. A VM is considered as reachable if it can be pinged. As it can be observed the VMs submission time decreases with an increasing number of GMs. More precisely, when Snooze is deployed with 1 GM, 500 VMs become reachable after approximately 11 minute. However, when Snooze is deployed with 8 GMs, the same number of VMs become reachable after 3 minutes. These results highlight the benefits of the hierarchical architecture and horizontal GM scaling for the reactivity of VM submission.

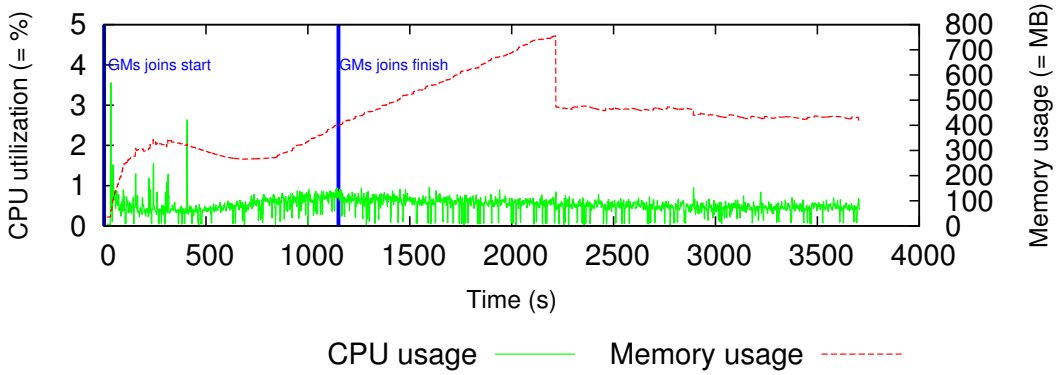
## 7.5 Scalability of the Self-configuration Mechanisms

In this section we evaluate the hierarchy construction time and demonstrate the upper bound on the number of GM and LC system services Snooze can manage. More precisely, we distinguish between three scenarios: (1) scalability of the GL; (2) scalability of a GM; (3) scalability of the whole hierarchy.

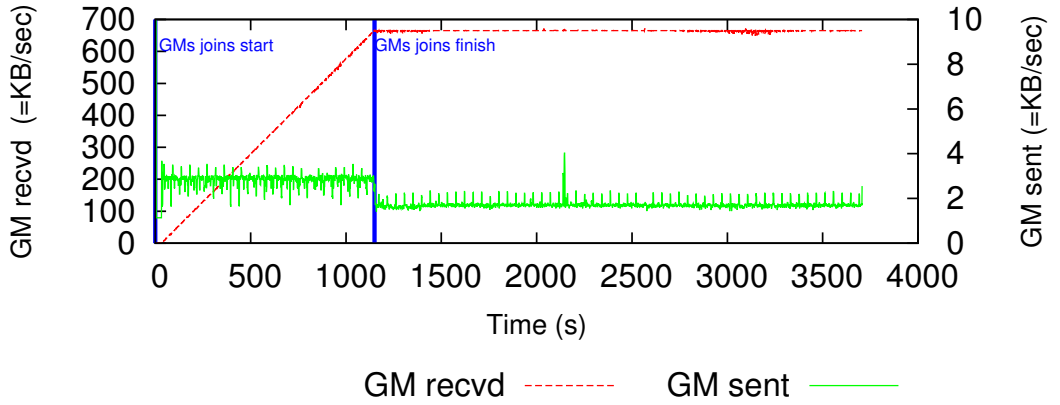
To evaluate the first scenario we deploy up to 5000 GMs on 100 servers with each server hosting 50 GM system services. The GL is hosted on a dedicated server. This server is started before the other servers to ensure that the GL election mechanism selects this servers as a leader. A similar experiment is conducted for the second scenario with up to 5000 LCs on 100 servers with each server hosting 50 LCs. We use one GM and one GL which are deployed on dedicated servers. In both experiments we measure the number of GMs and LCs which could join the system. In the third scenario we measure the time to construct the Snooze hierarchy with 1 GL, 1000 GMs (excluding the GL), and 10000 LCs. We use 20 servers hosting 50 GMs each, and 200 servers hosting 50 LCs each.

Table 1 depicts the number of system services which can successfully join the hierarchy in a given period of time. As it can be observed GMs join faster than LCs. This can be explained by the fact that the LC join procedure requires to contact both, the GL and the newly assigned GM. We observe that our prototype can handle up to approximately 4600 GMs and 4300 LCs per GL (resp. GM). Beyond this scale, the GL and GM system services start experiencing internal exceptions and networking timeouts. Finally, when considering the scalability of the whole hierarchy, all of the 1000 GMs and 10000 LCs can successfully join the hierarchy in 20 minutes. This evaluation shows that the self-configuration mechanisms used in Snooze is robust enough to deal with thousands of system services.

According to our experiments a GL and GM can handle approximately 4600 GMs (resp. 4300 LCs). Moreover, Snooze could easily manage a hierarchy of 1 GL, 1000 GMs, and 10000 LCs. Our results indicate that the system is able manage even larger hierarchies. However, due the resource limitations on the testbed we were not able to run larger experiments (i.e., millions of system services).




(a) GL CPU and memory consumption with an increasing number of GMs.



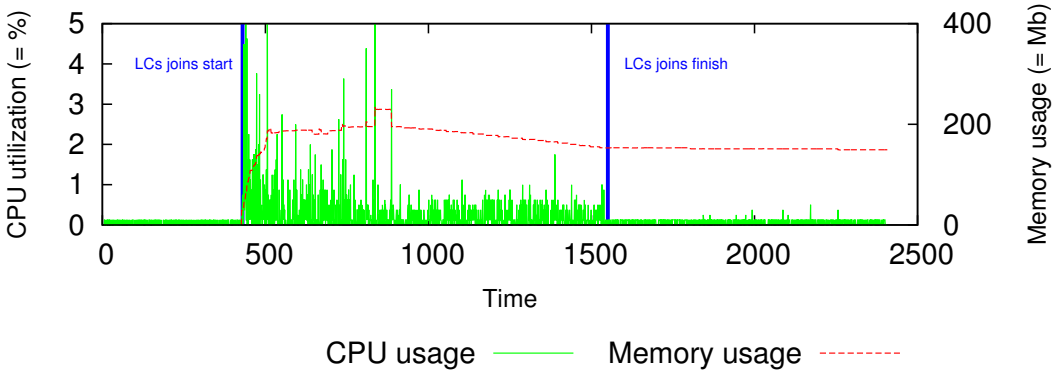

(b) GL network traffic with an increasing number of GMs.

Figure 5: GL CPU, memory, and network consummation with an increasing number of GMs. CPU consumption remains below 4%. Memory consumption stabilized at 450 MB. Network Rx traffic increases up to 650 KB/sec.

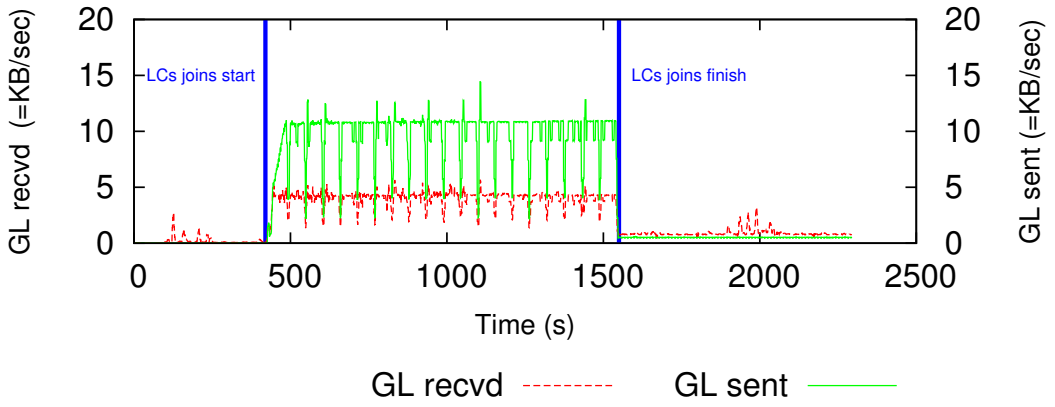
## 7.6 System Services Resource Consumption

In this section we show the resource consumption on the servers hosting the GL and GM system services. More precisely, we measure and report the CPU, memory, and network utilization. The measurements are done using the dstat tool [6]. We distinguish between two scenarios: (1) GL resource consumption with an increasing number of GMs and LCs; (2) GM resource consumption with an increasing number of LCs and VMs. Evaluating the GL resource consumption with an increasing number of GMs is especially important as the GL scalability directly depends on the number of managed GMs. Evaluating the GL resource consumption with an increasing number of LCs is especially important as the GL is in charge of assigning LCs to GMs during the LC join procedure. Evaluating the GM resource consumption with an increasing number of LCs and VMs is especially important as it demonstrates the upper bound on the number of server and VMs a GM can manage.

**Scenario 1: GL resource consumption with increasing numbers of GMs and LCs.** To evaluate the GL resource consumption with an increasing number of GMs, we deploy Snooze




(a) GL CPU and memory consumption with an increasing number of LCs.

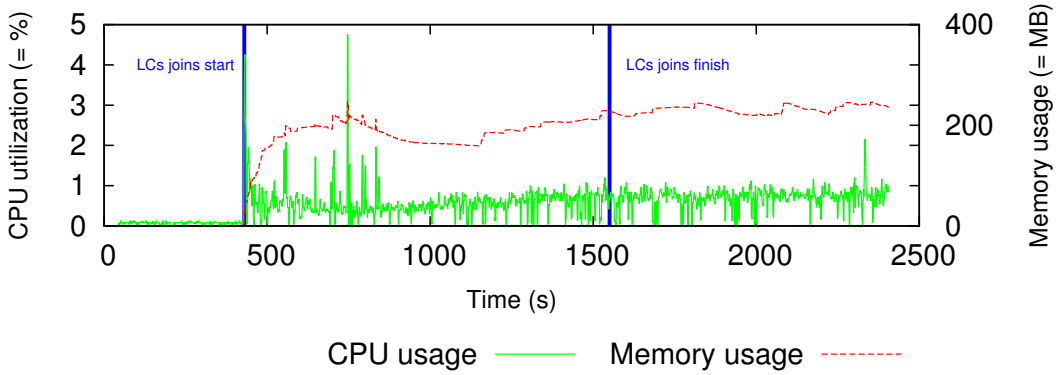



(b) GL network traffic with an increasing number of LCs.

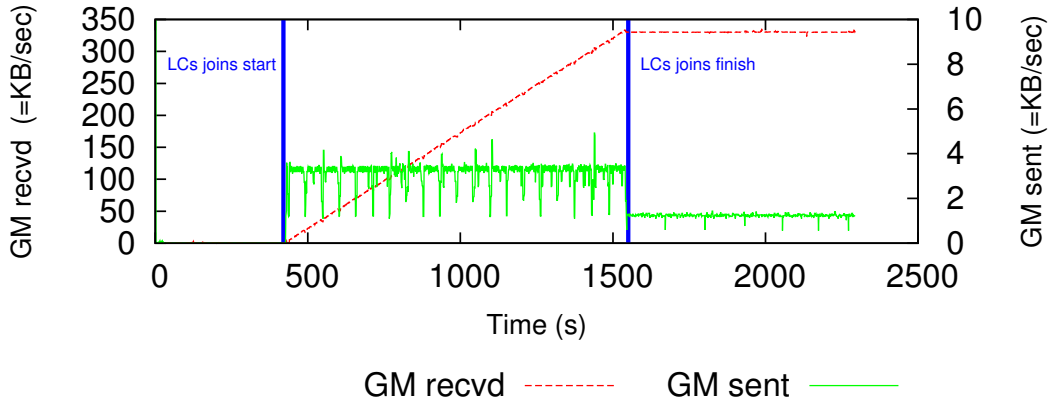
Figure 6: GL CPU, memory, and network consumption with an increasing number of LCs. CPU consumption reaches up to 5%. Memory consumption stabilizes at 180 MB. Network Rx traffic increases up to 10 KB/sec.

with one GL on a dedicated server. Moreover, we use 20 servers to host GM system services with each of them hosting 50 GMs resulting in 1000 GMs. We start GMs incrementally in one second intervals until 1000 GMs and measure the GL resource consumption. Note, that in this experiment no LCs are used as we are interested in the GL scalability with increasing number of GMs. The results from this evaluation are shown Figure 5. As it can be observed the CPU consumption remains below 4%. Memory consumption is proportional to the number of GMs in the system and reaches up to 450 MB. We believe that the large memory usage increase is related to the Snooze thread-based monitoring (i.e., one thread per GM) implementation which requires additional memory. The network Rx traffic is proportional to the number of GMs in the system due to the aggregated monitoring data sent by the GMs. The aggregated data size between one GM and the GL does not depends on the numbers of LCs or VMs in the system and is approximately 2 kB small. Sent traffic remains low since it corresponds to the heartbeat sent by the GL. GL heartbeats account to approximately 1 kB and thus are negligible at scale.

To evaluate the GL resource consumption with increasing number of LCs, we deploy Snooze with one GL and one GM. Both services are hosted on dedicated servers. We use 20 servers




(a) GM CPU and memory consumption with an increasing number of LCs.



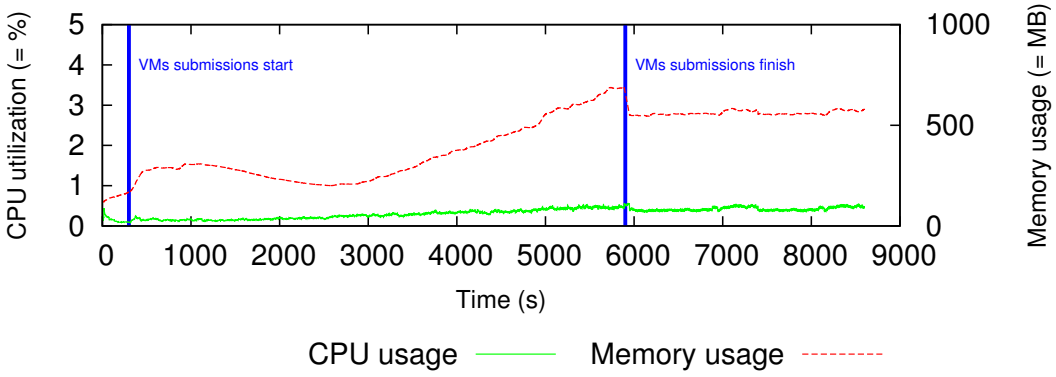

(b) GM network traffic with an increasing number of LCs.

Figure 7: GM CPU, memory, and network consumption with increasing number of LCs. CPU consumption stays below 5%. Memory consumption stabilizes at 220 MB. Network Rx traffic increases up to 300 KB/sec.

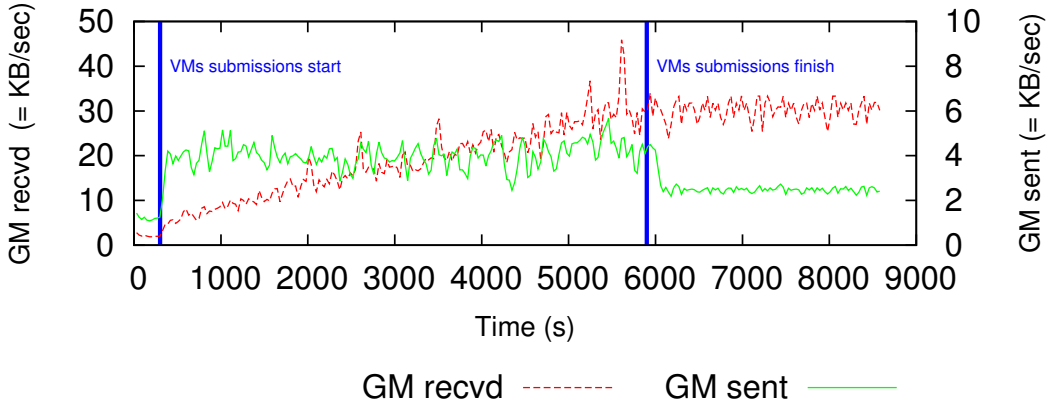
to serve as LCs with each of the servers hosting 50 LCs. We start LCs incrementally in one second intervals until 1000 LCs are reached. Figure 6 shows the GL resource consumption when increasing the number of LCs. As it can be observed CPU utilization spikes in the LC to GM assignment phase as a result of the LC assignment request processing. Memory stabilizes at approximately 180 MB. Network traffic remains low after the assignment phase. The memory and network consumption are the results of the GM aggregated data reception and GL heartbeat information sending.

**Scenario 2: GM resource consumption with increasing numbers of LCs and VMs.**

To evaluate the GM resource consumption with increasing number of LCs we measure the CPU, memory, and network utilization on the GM from the previous scenario. Figure 7 shows the results from this evaluation. During the LC assignment procedure, meta-data (i.e., IP and port) is first exchanged between the LC and GL (to get a GM assigned) as well as between the LC and the assigned GM (to join the GM). Only small CPU spikes can be observed during the LC join phase. No significant variation in CPU consumption is visible. Memory usage increases to about 220 MB. After the LC join phase, LC sends monitoring data to its assigned GM. As it




(a) GM CPU and memory consumption with an increasing number of VMs.




(b) GM network traffic with an increasing number of VMs.

Figure 8: GM CPU, memory, and network consumption with increasing number of VMs. CPU consumption increases by less than 1%. Memory consumption stabilizes at 500 MB. Network Rx traffic increases linearly.

can be observed, the GM network Rx traffic is proportional to the number of LCs whereas the Tx traffic remains low since only a fixed amount (i.e., 2 kB) of aggregated resource consumption data along with heartbeat messages is sent periodically to the GL.

To evaluate the resource consumption on the GM with an increasing number of VMs, we deploy Snooze with 1 GL, 1 GM and 20 LCs, each on a dedicated server and start up to 1000 VMs. VMs are configured with 1 VCORE, 256 MB of RAM, and 5 GB of disk space. We use QCOW2 disk images which are hosted on a Network File System (NFS). We start the VMs in groups of 10 incrementally up to 1000. Each LC hosts approximately 50 VMs. For this experiment the monitoring interval is set up to 6 seconds. The results are shown in Figure 8. As it can be observed, the CPU utilization remains under 1%. The memory usage increases up to 500 MB.

The GM Rx traffic increases proportionally with the numbers of VMs, while the send traffic remains low. Indeed, the Rx traffic is made of the monitoring data and the heartbeat sent by the LCs and the sent traffic consists in heartbeat packets sent by the GM. In the current prototype monitoring data is stored in-memory using a per VM ring buffer. Consequently, after some time

oldest monitoring values are overwritten resulting in a flat memory usage curve. The monitoring packet size between a LC and its assigned GM is approximately 1 kB plus 100 bytes per managed VM.

**Summary.** These experiments show that a GL can manage at least 1000 GMs. Moreover, each GM can manage at least 1000 LCs and 1000 VMs. Little CPU utilization spikes can be observed during the GM to GL join and LC to GM assignment processes. Memory consumption increases proportionally with the number of GMs, LCs, and VMs, as monitoring data is currently stored in-memory. Nevertheless, the total amount of memory used is limited by the size of the ring buffer set on each system service. Increasing the number of GMs and LCs increases the GL (resp. GM) network Rx traffic proportionally.

## 7.7 Scalability of the Self-healing Mechanisms

To evaluate the scalability of the self-healing mechanisms, our evaluation focuses on the hierarchy recovery time. We distinguish between two scenarios: (1) GL failure; (2) GM failures. Note that LC failures are out of the scope of this paper as the current Snooze prototype does not support the recovery of VM failures in the event of a LC crash. If the GL fails, the system is unable to accept new VM submission requests thus impacting the user experience. The time during which the system is unable to serve VM submission requests should be as short as possible. This is why we first measure the GL election time and investigate the impact of the number of GMs on the GL election time.

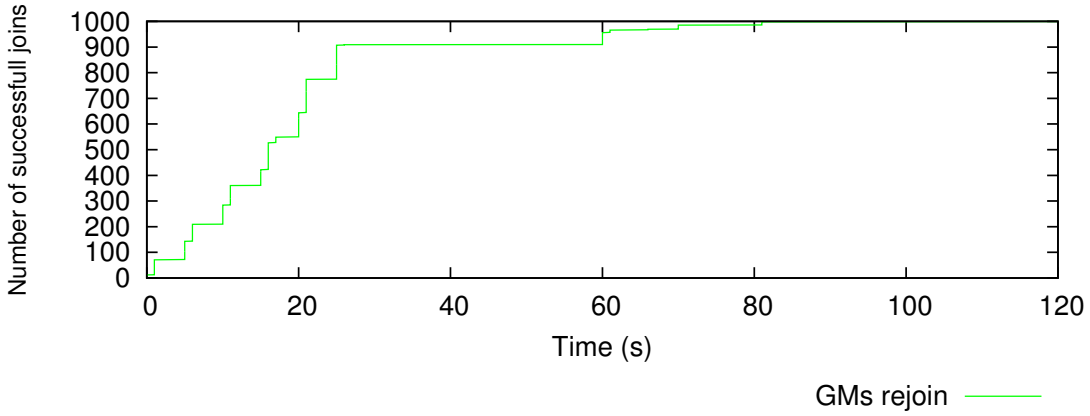
According to our measurements the GL election takes approximately 10 seconds and does not depend on the number of GMs. Moreover, in the event of a GL failure, GMs rejoin the newly elected GL in less than 2 minutes (see Figure 9a). Note that in this experiment we do not run any LC system services. However LCs are also impacted by a GL failure in the sense that the LCs of a newly promoted GM will have to rejoin the hierarchy. We cover this case in the second experiment which targets the recovery from GM failures.

To evaluate the hierarchy recovery time in the event of GM failures we deploy Snooze with 1 GL, 1000 GMs (excluding the GL), and 3000 LCs. We use 20 and 150 servers with each of them hosting 50 GMs and 20 LCs, respectively. This results in 3 LCs being assigned per GM. At arbitrary times, failures are injected by terminating the GM services on up to 19 servers, which represents 95% of the 20 servers hosting the GMs. We then measure the number of LCs on the remaining server hosting GM services. Figure 9b shows the GM failures as seen by the GL and the number of LCs rejoining the remaining GMs over time. As it can be observed all the LCs rejoin the hierarchy even when 950 GMs become unreachable in a short period of time. This result shows that Snooze can tolerate a large number of simultaneous GM failures. Indeed, after 20 minutes all LCs have rejoined the hierarchy.

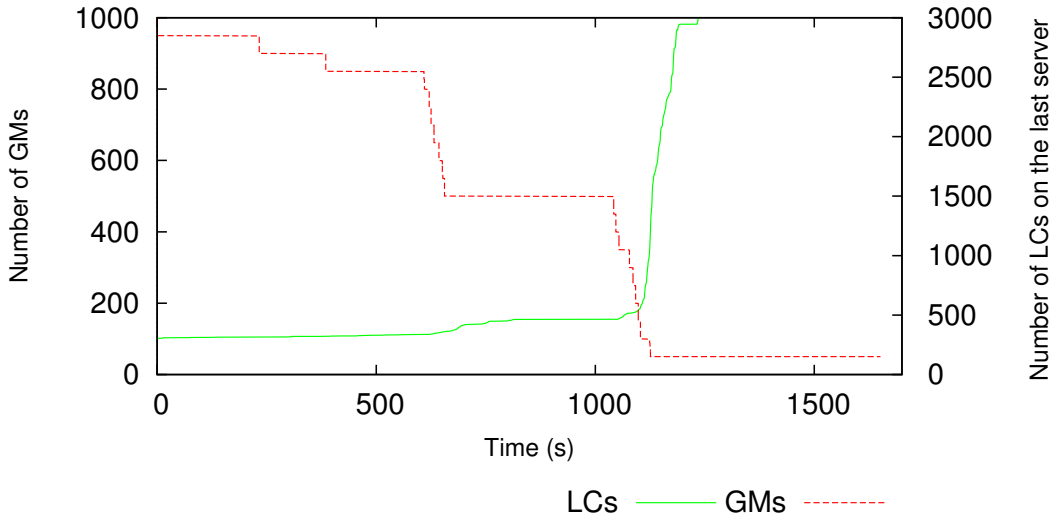
Finally, it is important to mention that if a system service at the management layer fails (e.g., a GM), all the system services assigned to it (e.g., all the LCs) trigger the hierarchy rejoin procedure almost simultaneously. In the event of simultaneous hierarchy rejoins, the GL becomes heavily loaded thus slowing down the rejoin time. The rejoin time can be reduced by tuning the threads pool of the HTTP server on the GL such that it can accept more concurrent connections. In our experiments, default Jetty HTTP server parameters are used [5].

## 7.8 Application Deployment Scalability and Impact of the Self-Healing Mechanisms

To evaluate the application deployment scalability and the impact of the self-healing mechanisms on the application performance, we have chosen Hadoop as it is the de-facto standard system for




(a) GMs rejoining after a GL failure.




(b) LCs rejoining after a large number of GM failures.

Figure 9: GMs and LCs rejoining after a GL failure (resp. a large number of concurrent GM failures). (a) GMs rejoining after a GL failure. A new GL is elected and all the remaining GMs rejoin the hierarchy within 80 seconds. (b) LCs rejoining in the event of GM failures. All LCs successfully rejoin the hierarchy after 20 minutes.

Big Data analytics in IaaS clouds. To conduct this experiment, Snooze is deployed with 1 EP, 1 GL, 8 GMs (excluding the GL), and 200 LCs each on a dedicated server. Our evaluation focuses on three aspects: (1) Hadoop VM submission; (2) Hadoop configuration; (3) Hadoop application execution and impact of the self-healing mechanisms.

Hadoop VM submission time captures the time to propagate VM images to the servers and the time for the VMs to become accessible. We use QCOW2 disk images which are propagated to the servers hosting the LCs using SCP Tsunami [8]. Hadoop v1.0.4 is pre-installed on the VMs. We then start 500 VMs (3 VCORES, 4 GB of RAM, and 50 GB of disk space each) in one step. After approximately 3 min, the VMs become available and Hadoop is configured using our

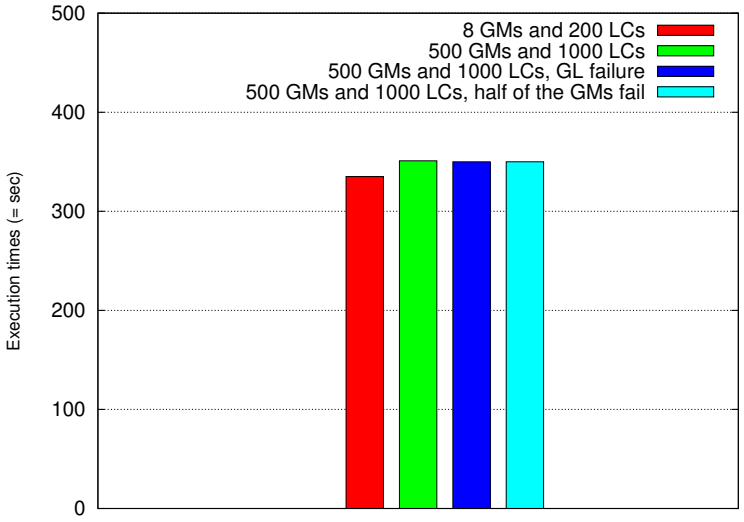



Figure 10: Impact of GL and GM failures on the TeraSort performance. A slight execution time increase can be observed with an increasing number of GMs. Self-healing mechanisms do not impact the performance.

scalable Hadoop configuration script with two maps and one reduce slots per VM. HDFS block size is set to 128 MB. The configuration takes 10 minutes.

To evaluate the impact of the self-healing mechanisms on the Hadoop execution time, we consider two kinds of failures: a GL failure and failure of half of the GMs. When a GL fails, all GMs need to rejoin the hierarchy. Moreover, the LCs which were assigned to a GM which has been promoted to a GL need to get a new GM assigned. Finally, when a GM failure occurs all its assigned LCs need to rejoin the hierarchy.

For the experiments we have selected the widely used TeraSort benchmark. We distinguish between three scenarios. In each scenario we measure the time taken to sort 100 GB of data: (1) TeraSort benchmark execution time on a Snooze deployment with 1 GL, 8 GMs, and 200 LCs; (2) we add GMs and LCs to the running cluster to reach 1 GL, 500 GMs and 1000 LCs and then execute the TeraSort benchmark; (3) we use the Snooze deployment from the second scenario and first inject a GL failure. We than inject a failure of half of the GMs. Failure injection is done randomly during the execution of the TeraSort benchmark. Each scenario is run three times and the average time taken by the TeraSort benchmark to sort the data is computed.

The results from this evaluation are shown in Figure 10. The first scenario achieves the best performance where TeraSort finishes in approximately 335 seconds. This can be explained with the fact that this scenario uses a small hierarchy of 8 GMs and 200 LCs thus resulting in low network contention. However, in the remaining two scenarios, the TeraSort benchmark finished in 351, 350 and 348 seconds, respectively. This is only approximately 4% higher than in the first scenario. These results show that hierarchy scale and maintenance have only negligible impact on the Hadoop performance.

# 8 Related Work

Several open-source as well as proprietary VM management frameworks such as OpenNebula [22], Nimbus [17], OpenStack [30], VMware vCenter Server [31], and Eucalyptus [23] have been developed during the last years. The architectures of the first four frameworks follow the traditional master-slave model and thus suffer from limited scalability and SPOF.

Similarly to Snooze, Eucalyptus is based on a hierarchical architecture. However, it relies on a static hierarchy and does not include any self-configuration and self-healing features.

In [27], a structured Peer-to-Peer (P2P)-based VM scheduling approach is introduced validated by simulation. However, this work does not implement any autonomy features. Moreover, no evaluation regarding its scalability and fault tolerance aspects is performed.

In [25], another structured P2P-based VM scheduling framework is proposed. The servers are organized in a ring and scheduling is performed iteratively upon underload and overload events. However, this system does not implement any self-organization or self-healing features. Moreover, the overheads of the ring maintenance and resource monitoring are not discussed.

In [21], the authors propose V-MAN, a VM management system based on an unstructured P2P-network of servers. In V-MAN, VM management decisions are applied only within the scope of randomly formed neighbourhoods. However, V-MAN does not implement self-configuration and healing features. Moreover, it has been validated by simulation only.

# 9 Conclusions and Future Work

In this paper, we have presented the design, implementation, and a large-scale evaluation of the Snooze autonomic IaaS cloud management system. Unlike the existing IaaS cloud management systems, Snooze is based on a self-configuring and healing hierarchy in which the VM management tasks are distributed across multiple self-healing group managers. Each group manager only maintains a partial view of the system. This allows the system to scale to thousands of servers and VMs. To the best of our knowledge this is the first work to present the internals of an autonomous IaaS cloud management system and perform a large-scale scalability evaluation in a realistic environment. Recent efforts (e.g., [28]) mainly focus on the functional aspects.

Our extensive evaluation shown that: (1) submission time is improved by performing distributed VM management; (2) the self-configuration mechanisms enable automated hierarchy construction in the presence of thousands of system services; (3) the server resource consumption scales well with an increasing number of system services; (4) the self-healing mechanisms enable hierarchy recovery in the presence of thousands of concurrent system services failures and do not impact application performance. Snooze is therefore suitable for managing large-scale virtualized data centers hosting thousands of servers. Finally, thanks to the integrated VM monitoring support as well as a modular design, the system is well suitable as a research testbed to experiment with advanced VM management strategies.

In the future, we plan to conduct a performance comparison of Snooze with existing open-source cloud management frameworks. Moreover, the Snooze system will be made even more autonomous by removing the distinction between GMs and LCs. Consequently, the decisions when a server should play the role of GM or LC in the hierarchy will be taken by the framework instead of the system administrator upon configuration. In addition, we will improve the scalability of the database and GL. Particularly, current in-memory repository implementation will be replaced by a distributed database. Scalability of the GL will be improved by adding replication and an additional load balancing layer. Finally, Snooze relies on the Network-File-System (NFS) as its VM disk image storage. Future work will investigate the use of scalable distributed file systems.

Snooze is an open-source software licensed under the GPL v2 license and available at `http://snooze.inria.fr`.

## Acknowledgments

This research was funded by the French Agence Nationale de la Recherche (ANR) project EcoGrappe and the Snooze advanced technological development action. Experiments presented in this paper were carried out using the Grid'5000 testbed (http://www.grid5000.fr), being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies.

# References

[1] MySQL Database. `http://www.mysql.com/`.

[2] Ganglia Monitoring System. `http://ganglia.sourceforge.net/`, 2011.

[3] Munin Network Monitoring Application. `http://www.munin-monitoring.org/`, 2012.

[4] ZooKeeper recipes and solutions. `http://zookeeper.apache.org/doc/trunk/recipes.html/`, 2012.

[5] Jetty parameters. `http://restlet.org/learn/javadocs/2.1/jee/engine/org/restlet/engine/connector/BaseHelper.html`, 2013.

[6] The dstat tool. `http://dag.wieers.com/home-made/dstat/`, 2013.

[7] The RENATER network. `http://www.renater.fr//`, 2013.

[8] The scp-tsnunami tool. `http://code.google.com/p/scp-tsunami/`, 2013.

[9] Inc. 10gen. MongoDB: A Scalable, High-Performance, Open Source NoSQL Database. `http://www.mongodb.org/`, 2012.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[11] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, 2005.

[12] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, 2009.

[13] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[14] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, 2011.

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, 2010.

[16] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning. *USENIX ;login:*, 38(1):38–44, February 2013.

[17] Kate Keahey, Tim Freeman, Jerome Lauret, and Doug Olson. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series*, 78(1):012038, 2007.

[18] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1), January 2003.

[19] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. In *Ottawa Linux Symposium*, July 2007.

[20] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a P2P network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009.

[21] Moreno Marzolla, Ozalp Babaoglu, and Fabio Panzieri. Server consolidation in Clouds through gossiping. In *Proceedings of the 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WOWMOM '11, 2011.

[22] Dejan Milojicic, Ignacio M. Llorente, and Ruben S. Montero. OpenNebula: A cloud management tool. *IEEE Internet Computing*, 15:11–14, March 2011.

[23] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[24] Srinath Perera and Dennis Gannon. Enforcing user-defined management logic in large scale systems. In *IEEE Congress on Services*, pages 243–250, 2009.

[25] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. Cooperative and reactive scheduling in large-scale virtualized platforms with dvms. *Concurrency and Computation: Practice and Experience*, 2012.

[26] Red Hat. libvirt: The virtualization API. `http://libvirt.org/`, 2012.

[27] Jonathan Rouzaud Cornabas. A distributed and collaborative dynamic load balancer for virtual machine. In *Proceedings of the 5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10) Euro-Par 2010*, Ischia, Naples Italy, 2010.

[28] P. Sempolinski and D. Thain. A comparison and critique of Eucalyptus, OpenNebula and Nimbus. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

[29] The Apache Software Foundation. CloudStack: Open Source Cloud Computing. `http://www.cloudstack.org/`, 2012.

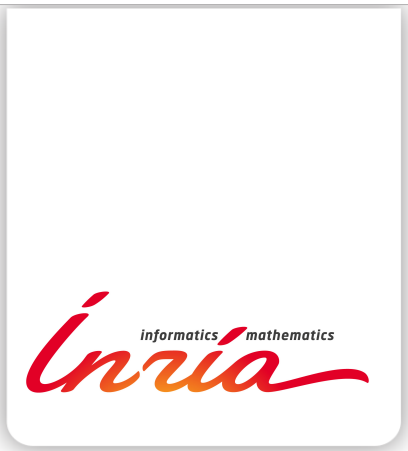
[30] The OpenStack Project. OpenStack: The Open Source Cloud Operating System. `http://www.openstack.org/software/`, 2012.

[31] VMware. Distributed Resource Management: Design, Implementation and Lessons Learned. `http://labs.vmware.com/publications/gulati-vmtj-spring2012`, 2012.

[32] Eric W Weisstein. L1-norm. `http://mathworld.wolfram.com/L1-Norm.html`, 2012.

[33] Katherine Yelick, Susan Coghlan, Brent Draney, and Richard S. Canon. The Magellan Report on Cloud Computing for Science. Technical report, U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research (ASCR), December 2011.

# Contents




**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399